Introduction to Neural Learning Gradient Descent The only relevant test of the validity of a hypothesis is comparison of prediction with experience. — MILTON FRIEDMAN 47 IN THIS CHAPTER Licensed to Asif Qamar Chapter 4 I 48 Introduction to Neural Learning Predict, Compare, and Learn This chapter is about "Compare", and "Learn" In Chapter 3, we learned about the paradigm: "Predict, Compare, Learn". In the previous chapter, we dove deep into the first part of this process "Predict". In this process we learned a myriad of things including the major parts of neural networks (nodes and weights), how datasets fit into networks (matching the number of datapoints coming in at one time), and finally how to use a neural network to make a prediction. Perhaps this process begged the question, "How do we set our weight values so that our network predicts accurately?". Answering this question will be the main focus of this chapter, covering the second two steps of our paradigm, "Compare", and "Learn". Compare A measurement of how much our prediction "missed". Once we've made a prediction, the next step to learn is to evaluate how well we did. Perhaps this might seem like a rather simple concept, but we will eventually find that coming up with a good way to measure error is one of the most important and complicated subjects of Deep Learning. In fact, there are many properties of "measuring error" that you have likely been doing your whole life without realizing it. Perhaps you (or someone you know) amplifies bigger errors while ignoring very small ones. In this chapter we will learn how to mathematically teach our network to do this. Furthermore (and this might seem too simple to be important), we will learn that error is always positive! We will consider the analogy of an "archer" hitting a target. Whether he is too low by and inch or too high by an inch, the error is still just 1 inch! In our neural network "Compare" step, we want to consider these kinds of properties when measuring error. As a heads up, in this chapter we will only evaluate one, very simple way of measuring error called "Mean Squared Error". However, it is but one of many ways to evaluate the accuracy of your neural network. As a closing thought, this step will give us a sense for "how much we missed", but this isn't enough to be able to learn. The output of our "compare" logic will simply be a "hot or cold" type signal. Given some prediction, we'll calculate an error measure that will either say "a lot" or "a little". It won't tell us why we missed, what direction we missed, or what we should do to fix it. It more or less just says "big miss", "little miss", or "perfect prediction". What we do about our error is captured in the next step, "Learn". Licensed to Asif Qamar Learn 49 Learn "Learning" takes our error and tells each weight how it can change to reduce it. Learning is all about "error attribution", or the art of figuring out how each weight played its part in creating error. It's the "blame game" of Deep Learning. In this chapter, we will spend a great number of pages learning the most popular version of the Deep Learning "blame game" called Gradient Descent. At the end of the day, it's going to result in computing a number for each of our weights. That number will represent how that weight should be higher or lower in order to reduce the error. Then we will move the weight according to that number, and we'll be done. Licensed to Asif Qamar Chapter 4 I 50 Introduction to Neural Learning Compare: Does our network make good predictions? Let's measure the error and fi nd out! Execute this code in your Jupyter notebook. It should print "0.3025". What is the goal_pred variable? Much like input, it's a number we recorded in the real world somewhere, but it's usu ally something that's hard to observe, like "the percentage of people who DID wear sweatsuits" given the temperature or "whether the batter DID in fact hit a home run" given his batting average. Why is the error squared? Th ink about an archer hitting a target. When he is 2 inches high, how much did he miss by? When he is two inches low, how much did he miss by? Both times he only missed by 2 inches. Th e primary reason why we square "how much we missed" is that it forces the output to be positive. pred-goal_pred could be negative in some situations... unlike actual error. Doesn't squaring make big errors (>1) bigger and small errors (Why measure error? 51 Why measure error? Measuring error simplifies the problem. The goal of training our neural network is to make correct predictions. That's what we want. And in the most pragmatic world (as mentioned in the last chapter), we want the network to take input

that we can easily calculate (today's stock price), and predict things that are hard to calculate (tomorrow's stock price). That's what makes a neural network useful. It turns out that "changing knob_weight to make the network correctly predict the goal_predic tion" is slightly more complicated than "changing the knob_weight to make error == 0" There's something more concise about looking at the problem this way. Ultimately, both of those state ments say the same thing, but trying to get the error to 0 just seems a bit more straightforward. Different ways of measuring error prioritize error differently. If this is a bit of a stretch right now, that's ok... but think back to what I said on the last page. By squaring the error, numbers that are less than 1 get smaller whereas numbers that are greater than 1 get bigger. This means that we're going to change what I call "pure error" (pre diction-goal_prediction) so that bigger errors become VERY big and smaller errors quickly become irrelevant. By measuring error this way, we can prioritize big errors over smaller ones. When we have somewhat large "pure errors" (say... 10), we're going to tell ourselves we have very large error (10**2 == 100), and in contrast, when we have small "pure errors" (say... 0.01), we're going to tell ourselves that we have very small error (0.01 **2 == 0.0001). See what I mean about prioritizing? It's just modifying what we consider to be error so that we amplify big ones and largely ignore small ones. In contrast, if we took the absolute value instead of squaring the error, we wouldn't have this type of prioritization. The error would just be the positive version of the "pure error"... which would be fine... just different. More on this later. Why do we only want positive error? Eventually, we're going to be working with millions of input -> goal_prediction pairs... and we're still going to want to make accurate predictions. This means that we're going to try to take the average error down to 0. This presents a problem if our error can be positive and negative. Imagine if we had two dat apoints... two input -> goal_prediction pairs that we were trying to get the neural network to correctly predict. If the first had an error of 1,000, and the second had an error of -1,000, then our average error would be ZERO! We would fool ourselves into thinking we predicted perfectly when we missed by 1000 each time!!! This would be really bad. Thus, we want the error of each prediction to always be positive so that they don't accidentally cancel each other out when we average them. Licensed to Asif Qamar Chapter 4 I 52 Introduction to Neural Learning What's the Simplest Form of Neural Learning? Learning using the Hot and Cold Method .1 input data enters here predictions come out here weight = 0.1 lr = 0.01 def neural_network(input, weight): prediction = input * weight return prediction 1 An Empty Network 2 PREDICT: Making A Prediction And Evaluating Error 8.5 0.85 number_of_toes = [8.5] win_or_lose_binary = [1] // (won!!!) input = number_of_toes[0] true = win_or_lose_binary[0] pred = neural_network(input,weight) error = (pred - true) ** 2 error #toes win? raw error Forces the raw error to be positive by multiplying it by itself. Negative error wouldn't make sense. .023 The "error" is simply a way of measuring "how much we missed". There are multiple ways to calculate error as we will learn later. This one is "Mean Squared Error" At the end of the day, learning is really about one thing, adjusting our knob_weight either up or down so that our error reduces. If we keep doing this and our error goes to 0, we are done learning! So, how do we know whether to turn the knob up or down? Well, we try both up and down and see which one reduces the error! Whichever one reduces the error is used to actually update the knob_weight. It's simple, but eff ective. Aft er we do this over and over again, eventually our error==0, which means our neural network is predicting with perfect accuracy. Wiggling our weights to see which direction reduces the error the most, moving our weights in that direction, and repeating until the error gets to 0. Hot and Cold Learning Licensed to Asif Qamar What's the Simplest Form of Neural Learning? 53 .09 4 COMPARE: Making A Prediction With a Lower Weight And Evaluating Error 8.5 0.85 lr = 0.01 p_dn = neural_network(input,weight-lr) .055 e_dn = (p_dn - true) ** 2 error .11 5 COMPARE + LEARN: Comparing our Errors and Setting our New Weight 8.5 0.85 if(error > e_dn || error > e_up): if(e_dn < e_up): weight -= lr if(e_up < e_up): weight += lr .055 errors .023 .004 down same up best!! .11 3 COMPARE: Making A Prediction With a Higher Weight And

Evaluating Error 8.5 0.85 lr = 0.01 p_up = neural_network(input,weight+lr) .004 e_up = (p_up - true) ** 2 We want to move the weight so that the error goes downward, so we're going to try moving the weight up and down to see which one has the lowest error. First, we're trying moving the weight up (weight+lr). error Th ese last 5 steps comprise 1 iteration of Hot and Cold Learning. Fortunately, this itera tion got us pretty close to the correct answer all by itself. (Th e new error is only 0.004). However, under normal circumstances, we would have to repeat this process many times in order to fi nd the correct weights. Some people even have to train their networks for weeks or months before they fi nd a good enough weight confi guration. Th is reveals what learning in neural networks really is. It's a search problem. We are searching for the best possible confi guration of weights so that our network's error falls to zero (and predicts perfectly). As with all other forms of search, we might not fi nd exactly what we're looking for, and even if we do, it may take some time. On the next page, we'll use Hot and Cold Learning for a slightly more diffi cult prediction so that you can see this searching in action! higher lower Licensed to Asif Qamar Chapter 4 I 54 Introduction to Neural Learning Hot and Cold Learning Perhaps the simplest form of learning. Execute this code in your Jupyter Notebook. (New neural network modifications are in bold.) This code attempts to correctly predict 0.8. weight = 0.5 input = 0.5 goal_prediction = 0.8 step_amount = 0.001 for iteration in range(1101): prediction = input * weight error = (prediction - goal_prediction) ** 2 print "Error:" + str(error) + " Prediction:" + str(prediction) up_prediction = input * (weight + step_amount) up_error = (goal_prediction - up_prediction) ** 2 down_prediction = input * (weight - step_amount) down_error = (goal_prediction - down_prediction) ** 2 if(down_error < up_error): weight = weight - step_amount if(down_error > up_error): weight = weight + step_amount Error:0.3025 Prediction:0.25 Error:0.30195025 Prediction:0.2505 .... Error:2.50000000033e-07 Prediction:0.7995 Error:1.07995057925e-27 Prediction:0.8 Our last step correctly predicts 0.8! TRY UP! TRY DOWN! If down is better, go down! If up is better, go up! how much to move our weights each iteration repeat learning many times so that our error can keep getting smaller When I run this code, I see the following output: Licensed to Asif Qamar Characteristics of Hot and Cold Learning 55 Characteristics of Hot and Cold Learning It's simple Hot and Cold learning is simple. After making our prediction, we predict two more times, once with a slightly higher weight and again with a slightly lower weight. We then move the weight depending on which direction gave us a smaller error. Repeating this enough times eventually reduces our error down to 0. PROBLEM #1: It's inefficient We have to predict multiple times in order to make a single knob_weight update. This seems very inefficient. PROBLEM #2: Sometimes it's impossible to predict the exact goal prediction. With a set step_amount, unless the perfect weight is exactly n*step_amount away, the network will eventually overshoot by some number less than step_amount. When it does so, it will then start alternating back and forth between each side of the goal_prediction. Set the step_amount to 0.2 to see this in action. If you set step_amount to 10 you'll really break it! When I try this I see the following output. It never remotely comes close to 0.8!!! Error:0.3025 Prediction:0.25 Error:19.8025 Prediction:5.25 Error:0.3025 Prediction:0.25 Error:19.8025 Prediction:5.25 Error:0.3025 Prediction:0.25 .... repeating infinitely... The real problem here is that even though we know the correct direction to move our weight, we don't know the correct amount. Since we don't know the correct amount, we just pick a fixed one at random (step_amount). Furthermore, this amount has NOTHING to do with our error. Whether our error is BIG What if we had a way of computing both direction and amount for each weight without having to repeatedly make predictions? or our error is TINY, our step_amount is the same. So, Hot and Cold Learning is kindof a bummer... it's inefficient because we predict 3 times for each weight update and our step_amount is completely arbitrary... which can prevent us from learning the correct weight value. Why did I iterate exactly 1101 times? The neural network reaches 0.8 after exactly that many iterations. If you go past that, it wiggles back and forth between 0.8 and just above/below 0.8... making for a less pretty error log printed at the bottom of

the left page. Feel free to try it out though. Licensed to Asif Qamar Chapter 4 I 56 Introduction to Neural Learning weight = 0.5 goal_pred = 0.8 input = 0.5 for iteration in range(20): pred = input * weight error = (pred - goal_pred) ** 2 direction_and_amount = (pred - goal_pred) * input weight = weight - direction_and_amount print "Error:" + str(error) + " Prediction:" + str(pred) Calculating Both direction and amount from error Let's measure the error and fi nd out! Execute this code in your Jupyter notebook. What you see above is a superior form of learning known as Gradient Descent. Th is method allows us to (in a single line of code... seen above in bold) calculate both the direction and the amount that we should change our weight so that we reduce our error. What is the direction_and_amount? It represents how we want to change our weight. Th e fi rst (1) is what we call "pure error" which equals (pred - goal_pred). Th is number represents "the raw direc tion and amount that we missed". Th e second part (2) is the multiplication by the input which performs scaling, negative reversal and stopping...modifying the "pure error" so that it's ready to update our weight. What is the "pure error"? It's the (pred - goal_pred) which indicates "the raw direction and amount that we missed". If this is a positive number then we predicted too high and vice versa. If this is a big number then we missed by a big amount, etc. What is "scaling, negative reversal, and stopping"? Th ese three attributes have the combined aff ect of translating our "pure error" into "the absolute amount that we want to change our weight". Th ey do so by addressing three major edge cases at which points the "pure error" is not suffi cient to make a good modifi cation to our weight. (1) "pure error" (2) scaling, negative reversal, and stopping 0.5 .30 .1 direction_and_amount -0.2 0.4 Licensed to Asif Qamar Calculating Both direction and amount from error 57 What is "stopping"? This is the first (and simplest) affect on our "pure error" caused by multiplying it by our input. Imagine plugging in a CD player into your stereo. If you turned the volume all the way up but the CD player was off... it simply wouldn't matter. "Stop ping" addresses this in our neural network... if our input is 0, then it will force our direction_and_amount to also be 0. We don't learn (i.e. "change the volume") when our input is 0 because there's nothing to learn... every weight value has the same error... and moving it makes no difference because the pred is always 0. What is "negative reversal"? This is probably our most difficult and important effect. Normally (when input is positive), moving our weight upward makes our prediction move upward. How ever, if our input is negative, then all of a sudden our weight changes directions!!! When our input is negative, then moving our weight up makes the prediction go down. It's reversed!!! How do we address this? Well, multiplying our "pure error" by our input will reverse the sign of our direction_and_amount in the event that our input is negative. This is "negative reversal", ensuring that our weight moves in the correct direction, even if the input is negative. What is "scaling"? Scaling is the second effect on our "pure error" caused by multiplying it by our input. Logically, it means that if our input was big, our weight update should also be big. This is more of a "side affect" as it can often go out of control. Later, we will use alpha to address when this scaling goes out of control. When you run the code in the top left, you should see the following output. Error:0.3025 Prediction:0.25 Error:0.17015625 Prediction:0.3875 Error:0.095712890625 Prediction:0.490625 ... Error:1.7092608064e-05 Prediction:0.79586567925 Error:9.61459203602e-06 Prediction:0.796899259437 Error:5.40820802026e-06 Prediction:0.797674444578 Our last steps correctly approach 0.8! In this example, we saw Gradient Descent in action in a bit of an oversimplified environment. On the next page, we're going to see it in it's more native environment. Some terminology will be different, but we will code it in a way that makes it more obviously applicable to other kinds of networks (such as those with multiple inputs and outputs) Licensed to Asif Qamar Chapter 4 I 58 Introduction to Neural Learning One Iteration of Gradient Descent This performs a weight update on a single "training example" (input->true) pair .1 input data enters here predictions come out here weight = 0.1 alpha = 0.01 def neural_network(input, weight): prediction = input * weight return prediction 1 An Empty Network 2 PREDICT: Making A Prediction And Evaluating Error 8.5 0.85 number_of_toes = [8.5]

win_or_lose_binary = [1] // (won!!!) input = number_of_toes[0] goal_pred = win_or_lose_binary[0] pred = neural_network(input,weight) error = (pred - goal_pred) ** 2 error #toes win? Forces the raw error to be positive by multiplying it by itself. Negative error wouldn't make sense. .023 The "error" is simply a way of measuring "how much we missed". There are multiple ways to calculate error as we will learn later. This one is "Mean Squared Error" 3 COMPARE: Calculating "Node Delta" and Putting it on the Output Node 8.5 .023 Delta is a measurement of "how much this node missed". Thus, since the true prediction was 1.0, and our network's prediction was 0.85, the network was too low by 0.15. Thus, delta is negative 0.15. -.15 number_of_toes = [8.5] win_or_lose_binary = [1] // (won!!!) input = number_of_toes[0] goal_pred = win_or_lose_binary[0] pred = neural_network(input,weight) error = (pred - goal_pred) ** 2 delta = pred - goal_pred node delta raw error Licensed to Asif Qamar One Iteration of Gradient Descent 59 4 LEARN: Calculating "Weight Delta" and Putting it on the Weight 8.5 .023 Weight delta is a measure of "how much this weight caused the newtork to miss". We calculate it by multiplying the weight's output "Node Delta" by the weight's input. Thus, we create each "Weight Delta" by scaling it's output "Node Delta" by the weight's input. This accounts for the 3 aforementioned properties of our "direction_and_amount", scaling, negative reversal, and stopping. .1 -.15 number_of_toes = [8.5] win_or_lose_binary = [1] // (won!!!) input = number_of_toes[0] goal_pred = win_or_lose_binary[0] pred = neural_network(input,weight) error = (pred - goal_pred) ** 2 delta = pred - goal_pred weight_delta = input * delta weight delta -1.25 .1125 5 LEARN: Updating the Weight number_of_toes = [8.5] win_or_lose_binary = [1] // (won!!!) input = number_of_toes[0] goal_pred = win_or_lose_binary[0] pred = neural_network(input,weight) error = (pred - goal_pred) ** 2 delta = pred - goal_pred weight_delta = input * delta alpha = 0.01 // fixed before training weight -= weight_delta * alpha We multiply our weight_delta by a small number "alpha" before using it to update our weight. This allows us to control how fast the network learns. If it learns too fast, it can update weights too aggressively and overshoot. More on this later. Note that the weight update made the same change (small increase) as Hot and Cold Learning new weight Th e primary diff erence between the gradient descent on the previous page and the im plementation on this page just happened. delta is a new variable. It's the "raw amount that the node was too high or too low". Instead of computing direction_and_amount directly, we fi rst calculate how much we wanted our output node to be diff erent. Only then do we compute our direction_and_amount to change the weight (in step 4, now renamed "weight_delta"). Licensed to Asif Qamar Chapter 4 I 60 Introduction to Neural Learning weight, goal_pred, input = (0.0, 0.8, 0.5) for iteration in range(4): pred = input * weight error = (pred - goal_pred) ** 2 delta = pred - goal_pred weight_delta = delta * input weight = weight - weight_delta print "Error:" + str(error) + " Prediction:" + str(pred) Learning Is Just Reducing Error Modifying weight to reduce our error. Putting together our code from the previous pages. We now have the following: these lines have a secret The Golden Method for Learning Adjusting each weight in the correct direction and by the cor rect amount so that our error reduces to 0. All we're trying to do is figure out the right direction and amount to modify weight so that our error goes down. The secret to this lies in our pred and error calculations. Notice that we actually use our pred inside the error calculation. Let's replace our pred variable with the code we used to generate it. error = ((input * weight) - goal_pred) ** 2 This doesn't change the value of error at all! It just combines our two lines of code so that we compute our error directly. Now, remember that our input and our goal_prediction are actually fixed at 0.5 and 0.8 respectively (we set them before the network even starts training). So, if we replace their variables names with the values... the secret becomes clear error = ((0.5 * weight) - 0.8) ** 2 Licensed to Asif Qamar Learning Is Just Reducing Error 61 error weight slope Th e Secret For any input and goal_pred, there is an exact relationship defi ned between our error and weight, found by combining our prediction and error formulas. In this case: error = ((0.5 * weight) - 0.8) ** 2 Let's say that you moved weight up by 0.5... if there is an exact

relationship between error and weight... we should be able to calculate how much this also moves the error! What if we wanted to move the error in a specifi c direction? Could it be done? Th is graph represents every value of error for every weight according to the relationship in the formula above. Notice it makes a nice bowl shape. Th e black "dot" is at the point of BOTH our current weight and error. Th e dotted "circle" is where we want to be (error == 0). Key Takeaway: Th e slope points to the bottom of the bowl (lowest error) no matter where you are in the bowl. We can use this slope to help our neural network reduce the error. Licensed to Asif Qamar Chapter 4 I 62 Introduction to Neural Learning error = 0.03 weight = 0.88 1.1 .03 0.17 .185 .97 Let's Watch Several Steps of Learning Will we eventually fi nd the bottom of the bowl? error = 0.64 weight = 0.0 1 1.1 .64 -.8 weight_delta = -0.88 (i.e. "raw error" modified for scaling, negative reversal, and stopping per this weight and input) -.88 0.0 delta (i.e. "raw error") 2 A Big Weight Increase Overshot a bit... Let's go back the other way weight, goal_pred, input = (0.0, 0.8, 1.1) for iteration in range(4): print "----- \nWeight:" + str(weight) pred = input * weight error = (pred - goal_pred) ** 2 delta = pred - goal_pred weight_delta = delta * input weight = weight - weight_delta print "Error:" + str(error) + " Prediction:" + str(pred) print "Delta:" + str(delta) + " Weight Delta:" + str(weight_delta) Licensed to Asif Qamar Let's Watch Several Steps of Learning 63 error = 0.000009 weight = 0.73 1.1 0.0000054 .007 .0081 .803 4 Ok, we're pretty much there... error = 0.002 weight = 0.69 1.1 .001 -.04 -.036 .76 3 Overshot Again! Let's go back again... but only just a little ----- Weight:0.0 Error:0.64 Prediction:0.0 Delta:-0.8 Weight Delta:-0.88 ----- Weight:0.88 Error:0.028224 Prediction:0.968 Delta:0.168 Weight Delta:0.1848 ----- Weight:0.6952 Error:0.0012446784 Prediction:0.76472 Delta:-0.03528 Weight Delta:-0.038808 ----- Weight:0.734008 Error:5.489031744e-05 Prediction:0.8074088 Delta:0.0074088 Weight Delta:0.00814968 Code Output Licensed to Asif Qamar Chapter 4 I 64 Introduction to Neural Learning Why does this work? What really is weight_delta? Let's back up and talk about functions. What is a function? How do we understand it? Consider this function: error = ((input * weight) - goal_pred) ** 2 def my_function(x): return x * 2 A function takes some numbers as input and gives you another number as output. As you can imagine, this means that the function actually defines some sort of relationship between the input number(s) and the output number(s). Perhaps you can also see why the ability to learn a function is so powerful... it allows us to take some numbers (say...image pixels) and convert them into other numbers (say... the probability that the image contains a cat). Now, every function has what you might call moving parts. It has pieces that we can tweak or change to make the ouput that the function generates different. Consider our "my_ function" above. Ask yourself, "what is controlling the relationship between the input and the output of this function?". Well, it's the 2! Ask the same question about the function below. What is controlling the relationship between the input and the output (error)? Well, plenty of things are! This function is a bit more complicated! goal_pred, input, **2, weight, and all the parenthesis and algebraic operations (addition, subtraction, etc.) play a part in calculating the error... and tweaking any one of them would change the error. This is import ant to consider. Just as a thought exercise, consider changing your goal_pred to reduce your error. Well, this is silly... but totally doable! In life, we might call this "giving up"... setting your goals to be whatever your capability is. It's just denying that we missed! This simply wouldn't do. What if we changed the input until our error went to zero... well... this is akin to seeing the world as you want to see it instead of as it actualy is. This is changing your input data until you're predicting what you want to predict (sidenote: this is loosely how "inceptionism works"). Now consider changing the 2... or the additions...subtractions... or multiplications... well this is just changing how you calculate error in the first place! Our error calculation is meaning less if it doesn't actually give us a good measure of how much we missed (with the right proper ties mentioned a few pages ago). This simply won't do either. Licensed to Asif Qamar Why does this work? What really is weight_delta? 65 So, what do we have left? The only variable we have left is our weight. Adjusting this doesn't change our perception of the world...

doesn't change our goal... and doesn't destroy our error measure. In fact, changing weight means that the function conforms to the patterns in the data. By forcing the rest of our function to be unchanging, we force our function to correctly model some pattern in our data. It is only allowed to modify how the network predicts. So, at the end of the day, we're modifying specific parts of an error function until the error value goes to zero. This error function is calculated using a combination of variables... some of them we can change (weights) and some of them we cannot (input data, output data, and the error logic itself). weight = 0.5 goal_pred = 0.8 input = 0.5 for iteration in range(20): pred = input * weight error = (pred - goal_pred) ** 2 direction_and_amount = (pred - goal_pred) * input weight = weight - direction_and_amount print "Error:" + str(error) + " Prediction:" + str(pred) We can modify anything in our pred calculation except the input. In fact, we're going to spend the rest of this book and many deep learning researchers will spend the rest of their lives just trying everything you can imagine to that pred calculation so that it can make good predictions. Learning is all about automatically changing that prediction function so that it makes good predictions... aka... so that the subsequent error goes down to 0. Ok, now that we know what we're allowed to change... how do we actually go about do ing that changing? That's the good stuff! That's the machine learning, right? In the next, section, we're going to talk about exactly that. Licensed to Asif Qamar Chapter 4 I 66 Introduction to Neural Learning Tunnel Vision on One Concept Concept: "Learning is adjusting our weight to reduce the error to zero" So far in this chapter, we've been hammering on the idea that learning is really just about adjusting our weight to reduce our error to zero. This is the secret sauce. Truth be told, knowing how to do this is all about understanding the relationship between our weight and our error. If we understand this relationship, we can know how to adjust our weight to reduce our error. What do I mean by "understand the relationship"? Well, to understand the relationship between two variables is really just to understand how changing one variable changes the other. In our case, what we're really after is the sensitivity between these two variables. Sensitivity is really just another name for direction and amount. We want to know how sensitive the error is to the weight. We want to know the direction and the amount that the error changes when we change the weight. This is the goal. So far, we've used two different methods to attempt to understand this relationship. You see, when we were "wiggling" our weight (hot and cold learning) and studying its affect on our error, we were really just experimentally studying the relationship between these two variables. It's like when you walk into a room with 15 different unlabeled light switches. You just start flipping them on and off to learn about their relationship to various lights in the room. We did the same thing to study the relationship between our weight and our error. We just wiggled the weight up and down and watched for how it changed the error. Once we knew the relationship, we could move the weight in the right direction using two simple if statements. if(down_error < up_error): weight = weight - step_amount if(down_error > up_error): weight = weight + step_amount Now, let's go back to the formula from the previous pages, where we combined our pred and error logic. As mentioned, they quietly define an exact relationship between our error and our weight. error = ((input * weight) - goal_pred) ** 2 This line of code, ladies and gentlemen, is the secret. This is a formula. This is the relationship between error and weight.This relationship is exact. It's computable. It's universal. It is and it will always be. Now, how can we use this formula to know how to change our weight so that our error moves in a particular direction. Now THAT is the right question! Stop. I beg you. Stop and appreciate this moment. This formula is the exact relationship between these two variables, and now we're going to figure out how to change one variable so that we move the other variable in a particular direction. As it turns out, there's a method for doing this for any formula. We're going to use it for reducing our error. Licensed to Asif Qamar A Box With Rods Poking Out of It 67 A Box With Rods Poking Out of It An analogy. Picture yourself sitting in front of a cardboard box that has two circular rods sticking through two little holes. The blue rod is sticking out of the box by 2 inches, and the red rod is sticking

out of the box by 4 inches. Imagine that I told you that these rods were connected in some way, but I wouldn't tell you in what way. You had to experiment to figure it out. So, you take the blue rod and push it in 1 inch, and watch as... while you're pushing... the red rod also moves into the box by 2 inches!!! Then, you pull the blue rod back out an inch, and the red rod follows again!!... pulling out by 2 inches. What did you learn? Well, there seems to be a relationship between the red and blue rods. However much you move the blue rod, the red rod will move by twice as much. You might say the following is true. As it turns out, there's a formal definition for "when I tug on this part, how much does this other part move". It's called a derivative and all it really means is "how much does rod X move when I tug on rod Y." In the case of the rods above, the derivative for "how much does red move when I tug on blue" is 2. Just 2. Why is it 2? Well, that's the multiplicative relationship determined by the formula. red_length = blue_length * 2 red_length = blue_length * 2 derivative Notice that we always have the derivative between two variables. We're always looking to know how one variable moves when we change another one! If the derivative is positive then when we change one variable, the other will move in the same direction! If the derivative is negative then when we change one variable, the other will move in the opposite direction. Consider a few examples. Since the derivative of red_length compared to blue_length is 2, then both numbers move in the same direction! More specifically, red will move twice as much as blue in the same direction. If the derivative had been -1, then red would move in the opposite direction by the same amount. Thus, given a function, the derivative represents the direction and the amount that one variable changes if you change the other variable. This is exactly what we were looking for! Licensed to Asif Qamar Chapter 4 I 68 Introduction to Neural Learning error = ((input * weight) - goal_pred) ** 2 Derivatives... take Two Still a little unsure about them?... let's take another perspective... Th ere are two ways I've heard people explain derivatives. One way is all about understanding "how one variable in a function changes when you move another variable". Th e other way of explaining it is "a derivative is the slope at a point on a line or curve". As it turns out, if you take a function and plot it out (draw it), the slope of the line you plot is the same thing as "how much one variable changes when you change the other". Let me show you by plotting our favorite function. Now remember... our goal_pred and input are fi xed, so we can rewrite this function: error = ((0.5 * weight) - 0.8) ** 2 Since there are only two variables left that actually change (all the rest of them are fi xed), we can just take every weight and compute the error that goes with it! Let's plot them error weight As you can see on the right, our plot looks like a big U shaped curve! Notice that there is also a point in the middle where the error == 0! Also notice that to the right of that point, the slope of the line is positive, and to the left of that point, the slope of the line is negative. Perhaps even more interesting, the farther away from the goal weight that you move, the steeper the slope gets. We like all of these properties. Th e slope's sign gives us direction and the slope's steepness gives us amount. We can use both of these to help fi nd the goal weight. starting "weight" weight = 0.5 error = 0.3025 direction_and_amount = -0.3025 goal "weight" weight = 1.6 error = 0.0 direction_and_amount = 0.0 Even now, when I look at that curve, it's easy for me to lose track of what it represents. It's actually similar to our "hot and cold" method for learning. If we just tried every possible value for weight, and plotted it out, we'd get this curve. And what's really remarkable about derivatives is that they can see past our big formula for computing error (at the top of this page) and see this curve! We can actually compute the slope (i.e. derivative) of the line for any value of weight. We can then use this slope (derivative) to fi gure out which direction reduces our error! Even better, based on the steepness we can get at least some idea for how far away we are (although not an exact one... as we'll learn more about later). slope Licensed to Asif Qamar What you really need to know... 69 What you really need to know... With derivatives... we can pick any two variables... in any formula... and know how they interact. Take a look at this big whopper of a function. Here's what you need to know about derivatives. For any function (even this whopper) you

can pick any two variables and understand their relationship with each other. For any func tion, you can pick two variables and plot them on an x-y graph like we did on the last page. For any function, you can pick two variables and compute how much one changes when you change the other. Thus, for any function, we can learn how to change one variable so that we can move another variable in a direction. Sorry to harp on, but it's important you know this in your bones. Bottom Line: In this book we're going to build neural networks. A neural network is really just one thing... a bunch of weights which we use to compute an error function. And for any error function (no matter how complicated), we can compute the relationship between any weight and the final error of the network. With this information, we can change each weight in our neural network to reduce our error down to 0... and that's exactly what we're going to do. y = (((beta * gamma) ** 2) + (epsilon + 22 - x)) ** (1/2) What you don't really need to know... ....Calculus.... So, it turns out that learning all of the methods for taking any two variables in any func tion and computing their relationship takes about 3 semesters of college. Truth be told, if you went through all three semesters so that you could learn how to do Deep Learning... you'd only actually find yourself using a very small subset of what you learned. And really, Calculus is just about memorizing and practicing every possible derivative rule for every possible function. So, in this book I'm going to do what I typically do in real life (cuz i'm lazy?... i mean... efficient?) ... just look up the derivative in a reference table. All you really need to know is what the derivative represents. It's the relationship between two variables in a function so that you can know how much one changes when you change the other. It's just the sensitivity between two variables. I know that was a lot of talking to just say "It's the sensitivity between two variables"... but it is. Note that this can include both "positive" sensitity (when variables move together) and "negative" sensitivity (when they move in opposite directions) or "zero" sensitivity...where one stays fixed regardless of what you do to the other. For example, y = 0 * x. Move x... y is always 0. Ok, enough about derivatives. Let's get back to Gradient Descent. Licensed to Asif Qamar Chapter 4 I 70 Introduction to Neural Learning error starting "weight" weight = 0.5 error = 0.3025 weight_delta = -0.3025 goal "weight" weight = 1.6 error = 0.0 weight_delta = 0.0 slope How to use a derivative to learn "weight_delta" is our derivative. What is the diff erence between the error and the derivative of our error and weight? Well the error is just a measure of how much we missed. Th e derivative defi nes the realtionship between each weight and how much we missed. In other words, it tells how much changing a weight contributed to the error. So, now that we know this, how do we use it to move the error in a particular direction? So, we've learned the relationship between two variables in a function... how do we ex ploit that relationship? As it turns out, this is incredibly visual and intuitive. Check out our error curve again. Th e black dot is where our weight starts out at (0.5). Th e dotted circle is where we want it to go... our goal weight. Do weight you see the dotted line attached to our black dot? Th at's our slope otherwise known as our derivative. It tells us at that point in the curve how much the error changes when we change the weight. Notice that it's pointed down ward! It's a negative slope! Th e slope of a line or curve always points in the opposite direction to the lowest point of the line or curve. So, if you have a negative slope, you increase your weight to fi nd the mini mum of the error. Check it out! So, how do we use our derivative to fi nd the error minimum (lowest point in the error graph)? We just move the opposite direction of the slope! We move in the opposite direc tion of the derivative! So, we can take each weight, calculate the derivative of that weight with respect to the error (so we're comparing two variables there... the weight and the error) and then change the weight in the opposite direction of that slope! Th at will move us to the minimum! Let's remember back to our goal again. We are trying to fi gure out the direction and the amount to change our weight so that our error goes down. A derivative gives us the relation ship between any two variables in a function. We use the derivative to determine the relationship between any weight and the error. We then move our weight in the opposite direction of the derivative to fi nd the lowest weight. Wallah! Our

neural network learns! Th is method for learning (fi nding error minimums) is called Gradient Descent. Th is name should seem intuitive! We move in the weight value opposite the gradient value, which descends our error to 0. By opposite, I simply mean that we increase our weight when we have a negative gradient and vice versa. It's like gravity! Licensed to Asif Qamar Look Familiar? 71 error = 0.03 weight = 0.88 1.1 .03 0.17 .187 .97 error = 0.64 weight = 0.0 1 1.1 .64 -.8 weight_delta = -0.88 (i.e. "raw error" modified for scaling, negative reversal, and stopping per this weight and input) -.88 0.0 delta (i.e. "raw error") 2 A Big Weight Increase Overshot a bit... Let's go back the other way Look Familiar? weight = 0.0 goal_pred = 0.8 input = 1.1 for iteration in range(4): pred = input * weight error = (pred - goal_pred) ** 2 delta = pred - goal_pred weight_delta = delta * input weight = weight - weight_delta print "Error:" + str(error) + " Prediction:" + str(pred) derivative (i.e., how fast the error changes given changes in the weight) Licensed to Asif Qamar Chapter 4 I 72 Introduction to Neural Learning Breaking Gradient Descent Just Give Me The Code weight = 0.5 goal_pred = 0.8 input = 0.5 for iteration in range(20): pred = input * weight error = (pred - goal_pred) ** 2 delta = pred - goal_pred weight_delta = input * delta weight = weight - weight_delta print "Error:" + str(error) + " Prediction:" + str(pred) Error:0.3025 Prediction:0.25 Error:0.17015625 Prediction:0.3875 Error:0.095712890625 Prediction:0.490625 ... Error:1.7092608064e-05 Prediction:0.79586567925 Error:9.61459203602e-06 Prediction:0.796899259437 Error:5.40820802026e-06 Prediction:0.797674444578 When I run this code, I see the following output... Now that it works... let's break it! Play around with the starting weight, goal_pred, and input numbers. You can set them all to just about anything and the neural network will figure out how to predict the output given the input using the weight. See if you can find some combinations that the neural network cannot predict! I find that trying to break some thing is a great way to learn about it. Let's try setting input to be equal to 2, but still try to get the algorithm to predict 0.8. What happens? Well, take a look at the output. Error:0.04 Prediction:1.0 Error:0.36 Prediction:0.2 Error:3.24 Prediction:2.6 ... Error:6.67087267987e+14 Prediction:-25828031.8 Error:6.00378541188e+15 Prediction:77484098.6 Error:5.40340687069e+16 Prediction:-232452292.6 Woah! That's not what we wanted! Our predictions exploded! They alternate from negative to positive and negative to positive, getting farther away from the true answer at every step! In other words, every update to our weight overcorrects! In the next section, we'll learn more about how to combat this phenomenon. Licensed to Asif Qamar Visualizing the Overcorrections 73 error = 3.24 weight = 1.3 2.0 3.24 1.8 3.6 2.6 3 Overshot Again! Let's go back again... but only just a little error = 0.36 weight = 0.1 2.0 .36 -.6 -1.2 0.2 error = 0.04 weight = 0.5 1 2.0 .04 0.2 weight_delta = -0.28 (i.e. "raw error" modified for scaling, negative reversal, and stopping per this weight and input) 0.4 1.0 delta (i.e. "raw error") 2 A Big Weight Increase Overshot a bit... Let's go back the other way Visualizing the Overcorrections Licensed to Asif Qamar Chapter 4 I 74 Introduction to Neural Learning Divergence Sometimes... neural networks explode in value... oops? derivative value start goal 1st step 2nd step 3rd step So what really happened? Th e explosion in error on the previous page is caused by the fact that we made the input larger. Consider how we're updating our weight. If our input is suffi ciently large, this can make our weight update large even when our error is small. What happens when you have a large weight update and a small error? It overcorrects!!! If the new error is even bigger, it overcorrects even more!!! Th is causes the phenomenon that we saw on the previous page, called divergence. You see, if we have a BIG input, then the prediction is very sensitive to changes in the weight (since pred = input * weight). Th is can cause our network to overcorrect. In other words, even though our weight is still only starting at 0.5, our derivative at that point is very steep. See how tight the u shaped error curve is in the graph above? Th is is actually really intuitive. How do we predict? Well, we predict by multiplying our input by our weight. So, if our input is huge, then small changes in our weight are go ing to cause BIG changes in our prediction!! Th e error is very sensitive to our weight. Aka... the derivative is really big! So, how do

we make it smaller? weight value weight = weight - (input * (pred - goal_pred)) Licensed to Asif Qamar Introducing.... Alpha 75 Introducing.... Alpha The simplest way to prevent overcorrecting our weight updates. So, what was the problem we're trying to solve? Th e problem is this: if the input is too big, then our weight update can overcorrect. What is the symptom? Th e symptom is that when we overcorrect, our new derivative is even larger in magnitude than when we started (although the sign will be the opposite). Stop and consider this for a second. Look at the graph above to understand the symptom. Th e 2nd step is even farther away from the goal... which means the derivative is even greater in magnitude! Th is causes the 3rd step to be even farther away from the goal than the second step, and the neural network continues like this, demonstrating divergence. Th e symptom is this overshooting. Th e solution is to multiply the weight update by a fraction to make it smaller. In most cases, this involves multiplying our weight update by a single real-valued number between 0 and 1, known as alpha. One might note, this has no af fect on the core issue which is that our input is larger. It will also reduce the weight updates for inputs that aren't too large. In fact, fi nding the appropriate alpha, even for state-of-the art neural networks, is oft en done simply by guessing. You watch your error over time. If it starts diverging (going up), then your alpha is too high, and you decrease it. If learning is happening too slowly, then your alpha is too low, and you increase it. Th ere are other meth ods than simple gradient descent that attempt to counter for this, but gradient descent is still very popular. derivative value start goal 1st step 2nd step 3rd step weight value Licensed to Asif Qamar Chapter 4 I 76 Introduction to Neural Learning Alpha In Code Where does our "alpha" parameter come in to play? So we just learned that alpha reduces our weight update so that it doesn't overshoot. How does this affect our code? Well, we were updating our weights according to the following formula. Accounting for alpha is a rather small change, pictured below. Notice that if alpha is small (say...0.01), it will reduce our weight update considerably, thus preventing it from over shooting. Well, that was easy! So, let's install alpha into our tiny implementation from the beginning of this chapter and run it where input = 2 (which previously didn't work) Error:0.04 Prediction:1.0 Error:0.0144 Prediction:0.92 Error:0.005184 Prediction:0.872 ... Error:1.14604719983e-09 Prediction:0.800033853319 Error:4.12576991939e-10 Prediction:0.800020311991 Error:1.48527717099e-10 Prediction:0.800012187195 Wallah! Our tiniest neural network can now make good predictions again! How did I know to set alpha to 0.1? Well, to be honest, I just tried it and it worked. And despite all the crazy advancements of deep learning in the past few years, most people just try several orders of magnitude of alpha (10,1,0.1,0.01,0.001,0.0001) and then tweak from there to see what works best. It's more art than science. There are more advanced ways which we can get to later, but for now, just try various alphas until you get one that seems to work pretty well. Play with it! What happens when you make alpha crazy small or big? What about making it negative? weight = weight - derivative weight = weight - (alpha * derivative) weight = 0.5 goal_pred = 0.8 input = 2 alpha = 0.1 for iteration in range(20): pred = input * weight error = (pred - goal_pred) ** 2 derivative = input * (pred - goal_pred) weight = weight - (alpha * derivative) print "Error:" + str(error) + " Prediction:" + str(pred) Licensed to Asif Qamar Memorizing 77 Memorizing Ok... it's time to really learn this stuff This may sound like something that's a bit intense, but I can't stress enough the value I have found from this exercise. The code on the previous page, see if you can build it in an iPy thon notebook (or a .py file if you must) from memory. I know that might seem like overkill, but I (personally) didn't have my click moment with neural networks until I was able to perform this task. Why does this work? Well, for starters, the only way to know that you have gleaned all the information necessary from this chapter is to try to produce it just from your head. Neural networks have lots of small moving parts, and it's easy to miss one. Why is this important for the rest of the chapters? In the following chapters, I will be re ferring to the concepts discussed in this chapter at a faster pace so that I can spend plenty of time on the newer material. It is vitally important that when I say something like

"add your alpha parameterization to the weight update" that it is at least immediately apparent to which concepts from this chapter I'm referring. All that is to say, memorizing small bits of neural network code has been hugely beneficial for me personally, as well as to many individuals who have taken my advice on this subject in the past.